CLEF, a Java Library to Extract Logical Relationships from Multivalued Contexts

Jessie Carbonnel

LIRMM, CNRS and Université de Montpellier, Montpellier, France
jcarbonnel@lirmm.fr

Abstract. Binary contexts and their associated conceptual structures embody several types of information that are true for the considered set of objects. Extracting binary implications and implicational bases received a lot of attention at first, but over the past years, notably in the domain of software reverse engineering, several methods and algorithms have been proposed to extract other kinds of information from conceptual structures, e.g., complex implications, mutual exclusions, prime implicants. In this paper, we present `CLEF`, a Java library implementing some of the literature algorithms for logical relationship extraction from conceptual structures. The library allows to process multivalued contexts with pattern structures and to automatically build a meet semi-lattice of patterns representing the similarities between objects having non-binary descriptions. The library is generic enough to extend it with (1) new extraction algorithms based on (2) different types of conceptual structures, and (3) new automated or semi-automated methods to build meet semi-lattices of patterns.

Keywords: Formal concept analysis, Pattern structures, Knowledge discovery, Multivalued contexts

1 Pattern Structures

Pattern structures [**?**] are a generalisation of formal concept analysis to characterise a set of objects O with more complex data than binary attributes. Therefore, objects of O may be described by multivalued contexts such as the one presented in Fig 1 (left). This excerpt presents a set of five objects (being versioning software systems) described by both binary and multivalued attributes.

In this generalised approach, each object is characterised by a *pattern* taken from a set of patterns (denoted D) having the same type. It is then necessary to define *similarities* between these patterns; the similarity of two patterns $d_1, d_2 \in D$ is given by a *similarity operator* (denoted $\sqcap$) that returns the most specific pattern (having the same type as d_1 and d_2) representing the similarity of d_1 and d_2. For instance, Kaytoue et al. [**?**] define the similarity of two intervals by the smallest interval containing them: [1986,1986] $\sqcap_{interv}$ [1998,1998] = [1986,1998],

Software	Merge	Lock	Program.Language	FirstRelease
Git	×		C; shell scripts; perl	2005
CVS	×		C	1986
ClearCase	×	×	C; java ; perl	/
GnuArch	×		C; shell scripts	2005
CVSNT	×	×	C++	1998

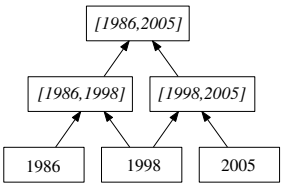

Fig. 1. (Left-hand side) Excerpt of a multivalued context about versioning software systems; (Right-hand side) Meet semi-lattice for the values of the attribute *FirstRelease*

with $\sqcap_{interv}$ the associated similarity operator. A subsumption relation $\sqsubseteq$ is associated to a similarity operator and partially orders the pattern set D by "specialisation": $a \sqsubseteq b \iff a \sqcap b = a, \forall a, b \in D$. The pair $(D, \sqcap)$ is a meet semi-lattice, i.e., a structure in which each subset of elements possesses an upper-bound representing their similarity. Figure 1 (right) presents the meet semi-lattice of values (patterns) taken from the column `FirstRelease` of Fig. 1 (left) and built with the interval similarity operator $\sqcap_{interv}$. The set of objects O, the meet semi-lattice of patterns $(D, \sqcap)$ and the mapping $\delta : O \rightarrow D$ that associates each object $o \in O$ with a pattern $d \in D$ form a pattern structure.

Patterns can be of atomic types (e.g., dates, numerical values, literals), but it is possible to combine several pattern types (from different pattern sets) in a *vector of patterns* [**?**]. A pattern vector is of the form $\langle d_1, d_2, \ldots, d_n \rangle$, where $d_i, i \in \{1, 2, \ldots, n\}$ is a pattern of the compound meet semi-lattice $(D_i, \sqcap_i)$. The similarity between two pattern vectors (denoted by $\sqcap_{pv}$) can be obtained by computing the similarity of patterns with the same rank in the vectors:

$$\langle d_1^k, d_2^k, ..., d_n^k \rangle \sqcap_{pv} \langle d_1^j, d_2^j, ..., d_n^j \rangle = \langle d_1^k \sqcap_1 d_1^j, d_2^k \sqcap_2 d_2^j, ..., d_n^k \sqcap_n d_n^j \rangle.$$

Therefore, one can automatically build a pattern structure from pattern vectors thanks to the compound meet semi-lattices. We estimate that pattern vectors are good candidates to handle objects described by several multivalued attributes, with potentially different types of values. From this kind of pattern structures, we can apply a binary scaling where each value of each compound meet semi-lattice becomes one binary attribute. The produced formal context can therefore be processed with traditional formal concept analysis algorithms and tools to build the conceptual structures associated with the scaled pattern structure. The obtained conceptual structures highlight the same information as the one found in the conceptual structures built without binary scaling.

2 Overview of the Java library `CLEF`

In what follows, we present the Java library `CLEF`[1] enabling to handle multivalued contexts, construct pattern vectors and their corresponding patterns structures,

[1] `https://gite.lirmm.fr/jcarbonnel/CLEF`

and extract information from the associated conceptual structures. It is composed of three packages: two focus on representing and managing multivalued contexts to be processed with pattern structures and formal concept analysis (Fig. 2) and one aims at extracting information from the conceptual structures (Fig. 3) obtained thank to the two previous packages.

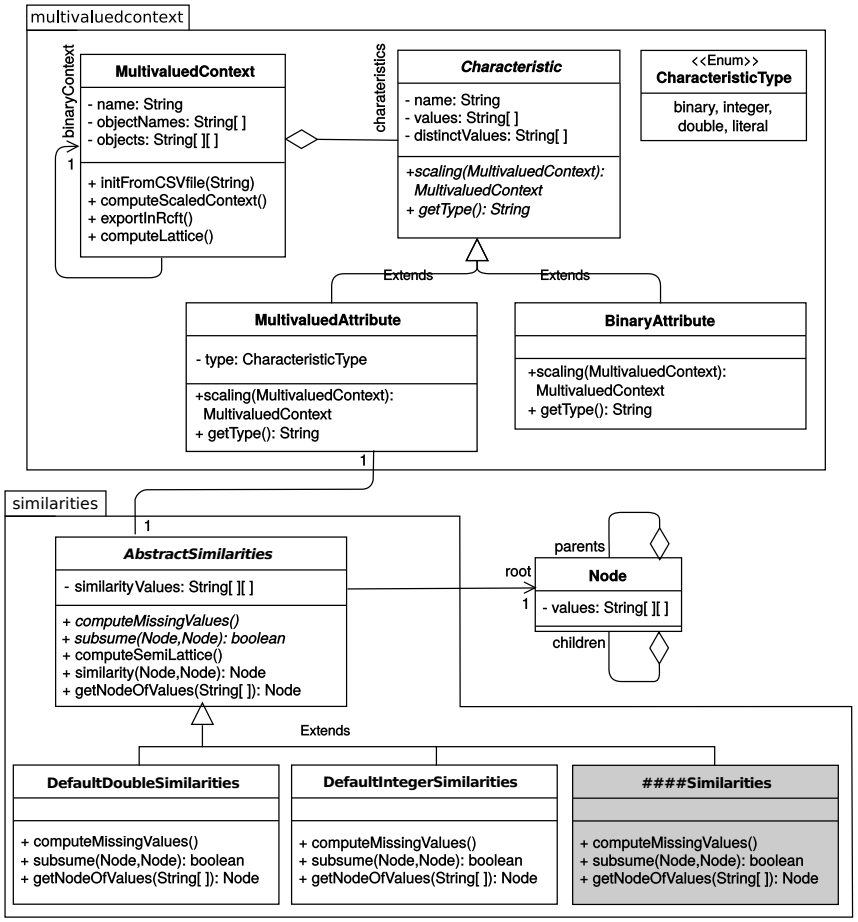

Fig. 2. UML class diagrams of `CLEF` (part 1)

Representing multivalued contexts (`multivaluedcontext`) The class `MultivaluedContext` is the core class of this package and represents objects, characteristics and their values. The class possesses one list of objects' names (e.g., [*Git*, *CVS*, *ClearCase*, *GnuArch*, *CVSNT*]) and one list of objects, an object being represented by the list of its values. For instance, the object *CVS* is represented by the list: [true, false, "C", 1986]. The class also has one list of `Characteristics`,

represented by an abstract class describing one column of the multivalued context and possessing a list of values (one per object). The class `Characteristic` has two sub-classes representing `BinaryAttribute`s (having boolean values) and `MultivaluedAttribute`s (having non-boolean values). We identified three types of values for `MultivaluedAttribute`s: "integer", "double" or "literal" (but this list can be extended). For instance, the characteristic *Lock* is a `BinaryAttribute` having the list of values [false, false, true, false, true], and the characteristic *FirstRelease* is a `MultivaluedAttribute` of type "integer" having the list of values [2005, 1986, /, 2005, 1998]. A characteristic has a method applying binary scaling on its values which produces a set of `BinaryAttribute`s. The scaling depends on the meet semi-lattice of values which are manage in the second package of Fig. 2 called `similarities`.

Building pattern structures (`similarities`) Each `MultivaluedAttribute` is associated with an instance of `AbstractSimilarities` structuring its set of (distinct) values depending on their similarities. As the set of distinct values may not be sufficient to build the meet semi-lattice of similarities, it is necessary to compute the missing values, i.e., the ones which are not present in the multivalued context. For instance, the three italic values in Fig. 1 (right) are "missing values" which are not present in the column *FirstRelease*. The construction of the meet semi-lattice (i.e., computing the missing values thanks to the similarity operator and organising the values by subsumption) depends on the type of values of the attribute as well as the user needs. Therefore, different strategies of construction may be implemented through sub-classes of `AbstractSimilarities`. We developed three strategies: `DefaultDoubleSimilarities` automatically builds a meet semi-lattice for values of type "double" based on the Sturges' rule; `DefaultIntegerSimilarities` automatically builds a meet semi-lattice for value of type "integer" using the similarity operator $\sqcap_{interv}$; `DefaultLiteralSimilarities` automatically builds a meet semi-lattice for values of type "literal" using formal concept analysis. If another strategy is needed, e.g., semi-automatically building a meet semi-lattice by asking the missing values to the user, one can add a new sub-class to `AbstractSimilarities`.

Applying binary scaling on a `MultivaluedAttribute` creates one `BinaryAttribute` for each value of the meet semi-lattice. Once the binary scaling applied on the whole multivalued context, it provides another `MultivaluedContext` having only binary attributes, i.e., a formal context. It can be saved in .RCFT format to be processed by the tool `RCAExplore`[2] to build the associate conceptual structures with traditional formal concept analysis algorithms.

Extracting relationships from conceptual structures (`relationshipextraction`) Concept lattices and their sub-hierarchies naturally highlight relationships between attribute values that can interest designers and domain experts. Binary implication extraction has received a lot of attention, partly because they are a compact and equivalent representation of the lattice and the formal context.

[2] `http://dataqual.engees.unistra.fr/logiciels`

Over the past years, notably in the domain of software reverse engineering, conceptual structures were used as a support to detect other kinds of information (e.g., complex implications, mutual exclusions, prime implicants) having in mind the same objective about compactly representing knowledge extracted from the conceptual structures and the context (i.e., extracting knowledge models).

The third package (Fig. 3) gathers classes implementing extraction algorithms. The class `AbstractACPosetExtractor` reads the information describing the AC-poset (i.e., the concept lattice sub-hierarchy limited to concepts introducing attributes) such as the concepts' intent and extent and their partial order. Extraction algorithms are then implemented through concrete sub-classes. For instance, we implemented a naive algorithm to extract all binary implications based on the AC-poset through `BinaryImplicationExtractor`, a sub-class of `AbstractACPosetExtractor`. We can imagine implementing a different algorithm by adding another dedicated sub-class. New abstract classes gathering information about other conceptual structures may also be added, for instance, a class `AbstractLatticeExtractor` with sub-classes implementing extraction algorithms based on concept lattices.

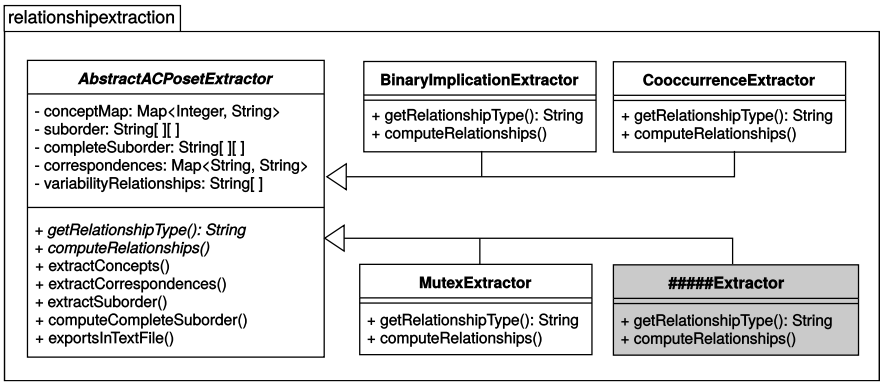

Fig. 3. UML class diagram of `CLEF` (part 2)

References

1. Ganter, B., Kuznetsov, S.O.: Pattern Structures and Their Projections. In: Proc. of the 9th Int. Conference on Conceptual Structures (ICCS'01). pp. 129–142 (2001)
2. Kaytoue, M., Kuznetsov, S.O., Napoli, A., Duplessis, S.: Mining gene expression data with pattern structures in formal concept analysis. Inf. Sci. 181(10), 1989–2001 (2011)
3. Kaytoue-Uberall, M., Assaghir, Z., Messai, N., Napoli, A.: Two complementary classification methods for designing a concept lattice from interval data. In: Proc. of the 6th Int. Symposium on Foundations of Information and Knowledge Systems (FoIKS'10). pp. 345–362 (2010)